

# Using cascading Bloom filters to improve the memory usage for de Bruijn graphs

Gregory Kucherov\*      Kamil Salikhov†

## Abstract

De Bruijn graphs are widely used in bioinformatics for processing next-generation sequencing data. Due to a very large size of NGS datasets, it is essential to represent de Bruijn graphs compactly, and several approaches to this problem have been proposed recently. In this note, we show how to reduce the memory required by the algorithm of [2] that represents de Bruijn graphs using Bloom filters. Our method requires 25% to 42% less memory with respect to the method of [2], with only insignificant increase in pre-processing and query times.

## 1 Introduction

Modern Next-Generation Sequencing (NGS) technologies generate huge volumes of short nucleotide sequences (*reads*) drawn from the DNA sample under study. The length of a read varies from 35 to about 400 basepairs (letters) and the number of reads may be hundreds of millions, thus the total volume of data may reach tens or even hundreds of Gb.

Many computational tools dealing with NGS data, especially those devoted to *genome assembly*, are based on the concept of *de Bruijn graph*, see e.g. [5]. The nodes of the de Bruijn graph are all distinct *k-mers* occurring

---

\*Université Paris-Est & CNRS, Laboratoire d'Informatique Gaspard Monge, Marne-la-Vallée, France, [Gregory.Kucherov@univ-mlv.fr](mailto:Gregory.Kucherov@univ-mlv.fr)

†Lomonosov Moscow State University, Moscow, Russia, [salikhov.kamil@gmail.com](mailto:salikhov.kamil@gmail.com)

in the reads, and two  $k$ -mers are linked by an edge if they occur at successive positions in a read<sup>1</sup>. In practice, the value of  $k$  is between 20 and 50.

The idea of using de Bruijn graph for genome assembly goes back to the “pre-NGS era” [6]. Note, however, that *de novo* assembly is not the only application of those graphs [4].

Due to a very large size of NGS datasets, it is essential to represent de Bruijn graphs compactly. Recently, several papers have been published that propose different approaches to compressing de Bruijn graphs [3, 7, 2, 1].

In this note, we focus on the method based on Bloom filters proposed in [2]. Bloom filters provide a very space-efficient representation of a subset of a given set (in our case, a subset of  $k$ -mers), at the price of allowing *one-sided errors*, namely *false positives*. The method of [2] is based on the following idea: if all queried nodes ( $k$ -mers) are only those which are reachable from some node known to belong to the graph, then only a fraction of all false positives can actually occur. Storing these false positives explicitly leads to an exact (false positive free) and space-efficient representation of the de Bruijn graph.

In this note we show how to improve this scheme by improving the representation of the set of false positives. We achieve this by iteratively applying a Bloom filter to represent the set of false positives, then the set of “false false positives” etc. We show analytically that this cascade of Bloom filters allows for a considerable further economy of memory, improving the method of [2]. Depending on the value of  $k$ , our method requires 25% to 42% less memory with respect to the method of [2]. Moreover, with our method, the memory grows very little with the growth of  $k$ . Finally, the pre-processing and query times increase only insignificantly compared to the original method of [2].

## 2 Cascading Bloom filter

Let  $T_0$  be the set of  $k$ -mers of the de Bruijn graph that we want to store. The method of [2] stores  $T_0$  via a bitmap  $B_1$  using a Bloom filter, together with the set  $T_1$  of *critical false positives*.  $T_1$  consists of those  $k$ -mers which are reachable from  $T_0$  by a graph edge and which are stored in  $B_1$  “by mistake”,

---

<sup>1</sup>Note that this actually a *subgraph* of the de Bruijn graph under its classical combinatorial definition. However, we still call it de Bruijn graph to follow the terminology common to the bioinformatics literature.

i.e. which belong to  $B_1$  but are not in  $T_0$ .<sup>2</sup>  $B_1$  and  $T_1$  are sufficient to represent the graph provided that the only queried  $k$ -mers are those which are adjacent to  $k$ -mers of  $T_0$ .

The idea we introduce in this note is to use this structure recursively and represent the set  $T_1$  by a new bitmap  $B_2$  and a new set  $T_2$ , then represent  $T_2$  by  $B_3$  and  $T_3$ , and so on. More formally, starting from  $B_1$  and  $T_1$  defined as above, we define a series of bitmaps  $B_1, B_2, \dots$  and a series of sets  $T_1, T_2, \dots$  as follows.  $B_2$  will store the set  $T_1$  of critical false positives using a Bloom filter, and the set  $T_2$  will contain “true nodes” from  $T_0$  that are stored in  $B_2$  “by mistake” (we call them **false**<sup>2</sup> positives).  $B_3$  and  $T_3$ , and, generally,  $B_i$  and  $T_i$  are defined similarly:  $B_i$  stores  $k$ -mers of  $T_{i-1}$  using a Bloom filter, and  $T_i$  contains  $k$ -mers stored in  $B_i$  “by mistake”, i.e. those  $k$ -mers that do not belong to  $T_{i-1}$  but belong to  $T_{i-2}$  (we call them **false**<sup>1</sup> positives). Observe that  $T_0 \cap T_1 = \emptyset$ ,  $T_0 \supseteq T_2 \supseteq T_4 \dots$  and  $T_1 \supseteq T_3 \supseteq T_5 \dots$ .

The following lemma shows that the construction is correct, that is it allows one to verify whether or not a given  $k$ -mer belongs to the set  $T_0$ .

**Lemma 1** *Given an element ( $k$ -mer)  $K$ , consider the smallest  $i$  such that  $K \notin B_{i+1}$  (if  $K \notin B_1$ , we define  $i = 0$ ). Then, if  $i$  is odd, then  $K \in T_0$ , and if  $i$  is even (including zero), then  $K \notin T_0$ .*

**Proof:** Observe that  $K \notin B_{j+1}$  implies  $K \notin T_j$  by the basic property of Bloom filters. We first check the Lemma for  $i = 0, 1$ .

For  $i = 0$ , we have  $K \notin B_1$ , and then  $K \notin T_0$ .

For  $i = 1$ , we have  $K \in B_1$  but  $K \notin B_2$ . The latter implies that  $K \notin T_1$ , and then  $K$  must be a false<sup>2</sup> positive, that is  $K \in T_0$ . Note that here we use the fact that the only queried  $k$ -mers  $K$  are either nodes of  $T_0$  or their neighbors in the graph (see [2]), and therefore if  $K \in B_1$  and  $K \notin T_0$  then  $K \in T_1$ .

For the general case  $i \geq 2$ , we show by induction that  $K \in T_{i-1}$ . Indeed,  $K \in B_1 \cap \dots \cap B_i$  implies  $K \in T_{i-1} \cup T_i$  (which, again, is easily seen by induction), and  $K \notin B_{i+1}$  implies  $K \notin T_i$ .

Since  $T_{i-1} \subseteq T_0$  for odd  $i$ , and  $T_{i-1} \subseteq T_1$  for even  $i$  (for  $T_0 \cap T_1 = \emptyset$ ), the lemma follows.  $\square$

Naturally, the lemma provides an algorithm to check if a given  $k$ -mer  $K$  belongs to the graph: it suffices to check successively if it belongs to

---

<sup>2</sup>By a slight abuse of language, we say that “an element belongs to  $B_j$ ” if it is accepted by the corresponding Bloom filter.

$B_1, B_2, \dots$  until we encounter the first  $B_{i+1}$  which does not contain  $K$ . Then the answer will simply depend on whether  $i$  is even or odd.

In our reasoning so far, we assumed an infinite number of bitmaps  $B_i$ . Of course, in practice we cannot store infinitely many (and even simply many) bitmaps. Therefore we “truncate” the construction at some step  $t$  and store a finite set of bitmaps  $B_1, B_2, \dots, B_t$  together with an explicit representation of  $T_t$ . The procedure of Lemma 1 is extended in the obvious way: if for all  $1 \leq i \leq t$ ,  $K \in B_i$ , then the answer is determined by directly checking  $K \in T_t$ .

### 3 Memory and time usage

First, we estimate the memory needed by our data structure, under the assumption of infinite number of bitmaps. Let  $N$  be the number of “true positives”, i.e. nodes of  $T_0$ . As it was shown in [2], if  $N$  is the number of nodes we want to store through a bitmap  $B_1$  of size  $rN$ , then the expected number of critical false positive nodes (set  $T_1$ ) will be  $8Nc^r$ , where  $c = 0.6185$ . Then, to store these  $8Nc^r$  critical false positive nodes, we use a bitmap  $B_2$  of size  $8rNc^r$ . Bitmap  $B_3$  is used for storing nodes of  $T_0$  which are stored in  $B_2$  “by mistake” (set  $T_2$ ). We estimate the number of these nodes as the fraction  $c^r$  (false positive rate of filter  $B_2$ ) of  $N$  (size of  $T_0$ ), that is  $Nc^r$ . Similarly, the number of nodes we need to put to  $B_4$  is  $8Nc^r$  multiplied by  $c^r$ , i.e.  $8Nc^{2r}$ . Keeping counting in this way, we obtain that the memory needed for the whole structure is  $rN + 8rNc^r + rNc^r + 8rNc^{2r} + rNc^{2r} + \dots$  bits. By dividing this expression by  $N$  to obtain the number of bits per one  $k$ -mer, we get  $r + 8rc^r + rc^r + 8rc^{2r} + \dots = r(1 + c^r + c^{2r} + \dots) + 8rc^r(1 + c^r + c^{2r} + \dots) = (1 + 8c^r) \frac{r}{1 - c^r}$ . A simple calculation shows that the minimum of this expression is achieved when  $r = 6.299$ , and then the minimum memory used per  $k$ -mer is 9.18801 bits.

As mentioned earlier, in practice we store only a finite number of bitmaps  $B_1, \dots, B_t$  together with an explicit representation (hash table) of  $T_t$ . In this case, the memory taken by the bitmaps is a truncated sum  $rN + 8rNc^r + rNc^r + \dots$ , and a hash table storing  $T_t$  takes either  $2k \cdot Nc^{\lceil \frac{t}{2} \rceil r}$  or  $2k \cdot 8Nc^{\lceil \frac{t}{2} \rceil r}$  bits, depending on whether  $t$  is even or odd. The latter follows from the observations that we need to store  $Nc^{\lceil \frac{t}{2} \rceil r}$  (or  $8Nc^{\lceil \frac{t}{2} \rceil r}$ )  $k$ -mers, and every  $k$ -mer takes  $2k$  bits of memory. Consequently, we have to adjust the optimal value of  $r$  minimizing the total space, and re-estimate the resulting space

$k$	optimal $r$	bits per $k$ -mer
16	6.447053	9.237855
32	6.609087	9.298095
64	6.848718	9.397210
128	7.171139	9.548099

Table 1: Optimal value of  $r$  (bitmap size of a Bloom filter per number of stored elements) and the resulting space per  $k$ -mer for  $t = 4$ .

spent on one  $k$ -mer.

Table 1 shows the optimal  $r$  and the space per  $k$ -mer for  $t = 4$  and several values of  $k$ . It demonstrates that even this small  $t$  leads to considerable memory savings. It appears that the space per  $k$ -mer is very close to the “optimal” space (9.18801 bits) obtained for the infinite number of filters. Table 1 reveals another advantage of our improvement: the number of bits per stored  $k$ -mer remains almost constant for different values of  $k$ .

We now compare the memory usage of our method compared to the original method of [2]. The data structure of [2] has been estimated to take  $(1.44 \log_2(\frac{16k}{2.08}) + 2.08)$  bits per  $k$ -mer. With this formula, comparative estimations of space consumption per  $k$ -mer by different methods are shown in Table 2. Observe that according to the method of [2], doubling the value of  $k$  results in a memory increase by a factor of 1.44, whereas in our method the increase is very small, as we already mentioned earlier.

$k$	“Optimal” (infinite) Cascading Bloom Filter	Cascading Bloom Filter with $t = 4$	Data structure from [2]
16	9.18801	9.237855	12.0785
32	9.18801	9.298095	13.5185
64	9.18801	9.397210	14.9585
128	9.18801	9.548099	16.3985

Table 2: Comparison of space consumption per  $k$ -mer for the “optimal” (infinite) cascading Bloom filter, finite ( $t = 4$ ) cascading Bloom filter, and the Bloom filter from [2], for different values of  $k$ .

Let us now estimate preprocessing and query times for our scheme. If the value of  $t$  is small (such as  $t = 4$ , as in Table 1), the preprocessing time grows insignificantly in comparison to the original method of [2]. To construct each  $B_i$ , we need to store  $T_{i-2}$  (possibly on disk, if we want to save on the internal memory used by the algorithm) in order to compute those  $k$ -mers which are stored in  $B_{i-1}$  “by mistake”. The preprocessing time increases little in comparison to the original method of [2], as the size of  $B_i$  decreases exponentially and then the time spent to construct the whole structure is linear on the size of  $T_0$ .

In the case of small  $t$ , the query time grows insignificantly as well, as a query may have to go through up to  $t$  Bloom filters instead of just one. The above-mentioned exponential decrease of sizes of  $B_i$  results in that the *average* query time will remain almost the same.

## References

- [1] Alexander Bowe, Taku Onodera, Kunihiko Sadakane, and Tetsuo Shibuya. Succinct de bruijn graphs. In Benjamin J. Raphael and Jijun Tang, editors, *Algorithms in Bioinformatics - 12th International Workshop, WABI 2012, Ljubljana, Slovenia, September 10-12, 2012. Proceedings*, volume 7534 of *Lecture Notes in Computer Science*, pages 225–235. Springer, 2012.
- [2] Rayan Chikhi and Guillaume Rizk. Space-efficient and exact de Bruijn graph representation based on a Bloom filter. In Benjamin J. Raphael and Jijun Tang, editors, *Algorithms in Bioinformatics - 12th International Workshop, WABI 2012, Ljubljana, Slovenia, September 10-12, 2012. Proceedings*, volume 7534 of *Lecture Notes in Computer Science*, pages 236–248. Springer, 2012.
- [3] Thomas C. Conway and Andrew J. Bromage. Succinct data structures for assembling large genomes. *Bioinformatics*, 27(4):479–486, 2011.
- [4] Z. Iqbal, M. Caccamo, I. Turner, P. Flicek, and G. McVean. De novo assembly and genotyping of variants using colored de Bruijn graphs. *Nat. Genet.*, 44(2):226–232, Feb 2012.
- [5] J. R. Miller, S. Koren, and G. Sutton. Assembly algorithms for next-generation sequencing data. *Genomics*, 95(6):315–327, Jun 2010.

- [6] P. A. Pevzner, H. Tang, and M. S. Waterman. An Eulerian path approach to DNA fragment assembly. *Proc. Natl. Acad. Sci. U.S.A.*, 98(17):9748–9753, Aug 2001.
- [7] Chengxi Ye, Zhanshan Ma, Charles Cannon, Mihai Pop, and Douglas Yu. Exploiting sparseness in de novo genome assembly. *BMC Bioinformatics*, 13(Suppl 6):S1, 2012.